NASA Workflow Tool

Tool Developer Guide

September 29, 2010

---

1. Overview

This document covers building and extending the NED related software. It is intended for those seeking to build and extend the tool software or create additional plug-ins for it.

2. Building the NED and LWWE Software

All the software is contained within the Progress SVN repository:

```
progress.nccs.nasa.gov/svn/workflowTool
```

The project is comprised of multiple Eclipse projects, and should be straightforward to build from within Eclipse.

The NED software packages contain ant scripts that are designed to automatically build “release” directories that can then be zipped up and distributed. Each distributable package contains all the necessary libraries it needs.

IMPORTANT: The JAVA_HOME environment variable is set correctly and that the java binaries are added to your path for proper compilation and operation.

These are necessary instructions for setting up the NED project. Download and install Eclipse with the Java Development tools (J2EE package).

Useful plugins for eclipse to download an install:

1. XJC (you must have JAXB for this to work)
2. Groovy Plugin
3. Jython/Python Plugin

You will need to perform a checkout from SourceMotel to retrieve the eclipse project and sources. This can be done through the eclipse GUI using Subclipse, for example.

Set the JAVA_LIB variable under project properties to point to where NEDLibs is, and ECLIPSE_HOME to where your Eclipse installation directory is, by going to: Eclipse->Preferences->Java->Build Path->Classpath Varibles. The other way to define these variables is to import from the file pref.epf under workflow/NEDLibs. The Eclipse command to do this: File->Import->General->Preferences, pick the reference file pref.epf, choose Import all, and click Finish. This preference file may need to be modified so the variables ECLIPSE_HOME points to where Eclipse is installed (the default value is for MAC OS).

You might need to go to: Project->Properties->Java Build Path to verify that all the paths there are correctly set to your local environment.

Additional Problems Discovered Trying to Build:

- Needed to fix the Builders in the project properties (delete broken one, add new one)
- Needed to fix the Targets in some of the Builders
- Some test folders needed to be created

3. Workflow Software Components

Because the NED tools are developed as component software, elements can be replaced with other elements. Internal software packages that can be used in other applications:

- LWWE Workflow Engine: This is the workflow engine that can be embedded in other applications.
- LWWE Workflow Engine Server: A standalone workflow engine server that can be used as a starting point for other servers.
- LWWE Workflow Client: A simple client that can interact with the LWWE workflow server. This can be extended or included in other applications.
- Generic Client/Generic Server: These are application skeletons that provide the basics needed for a client server application. These are used as the basis for the client applications (NED Client, NED Server Manager, LWWE Client) and server applications (NED Server, LWWE Server)

Along with these packages, the NED suite contains some useful libraries for development.

4. Plug-In Development

I. Introduction

NOTE: Client and server side hot pluggable functionality is not yet implemented. While the client side could do it with activities the reload the GUI, the server side does not. This functionality is in the workflow backlog.

The NED architecture is designed to allow for plug-in extensions. The mechanism for both client and server side extensions aimed for simplicity.

Here is a list of fundamental classes for attempting to write extensions to the NED interface. On the client side:

· ClientComponentInterface: The basic interface for any pluggable component.
· ConfigurationEditorInterface: The basic interface for custom editors within the NED GUI (extends from ClientComponentInterface).

For either type of plug-in, create an XML file like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ComponentConfiguration xmlns="NED">
        <Name>Component Name</Name>
        <Description>Description of component</Description>
        <Version>1.0.0.0</Version>
        <ConfigurationFile>CompConfig.xml</ConfigurationFile>
        <FileName>component.jar</FileName>
        <ClassName>component.componentclass </ClassName>
        <Type>JAVA</Type>
        <LinkedComponents></LinkedComponents>
</ComponentConfiguration>
```

This is a description of what each field does:

· Name: The name of the component.
· Description (optional): The description of the component
· Version (optional): The version of the component
· ConfigurationFile: The name of this configuration file.
· FileName: The filename of this component. This can be a class file, jar file, jython file, or groovy file.
· ClassName: The name of the class that derived from ClientComponentInterface or one of

its children.
· Type: The type of the component. This can be JAVA, JYTHON, or GROOVY
· LinkedComponents: Any dependencies this component has. Dependencies will be loaded before this component.

II. A Basic Component

So a completely basic client component class would look like this:

```
public class MyComponent implements ClientComponentInterface
{
public void initialize()
{
        System.out.println("I'm initializing");
}

        public void finish()
{
        System.out.println("I'm done.");
}
}
```

With an XML descriptor file MyComponent.xml like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ComponentConfiguration xmlns="NED">
        <Name>MyComponent</Name>
        <ConfigurationFile>MyComponent.xml</ConfigurationFile>
        <FileName>MyComponent.class</FileName>
        <ClassName>MyComponent</ClassName>
        <Type>JAVA</Type>
</ComponentConfiguration>
```

The component needs to have a constructor that takes no arguments. The initialize method is called AFTER the component has been constructed. DO NOT CALL THE INITIALIZE METHOD FROM THE CONSTRUCTOR. The client application manages this.

The finish method is called on application refreshes. This is important if you plan on registering objects with your component in your initialize method. The initialize method gets called again on refreshes so if you have created/registered any objects or listeners inside the initialize method they will get registered again.

A client component can be pretty much anything. For example, all core NED components are client components that provide anything from authentication dialogs to database operations.

When you want to add the component to NED, you can either place the class and the XML file in your NED client components directory or you can jar it up and place the jar in that directory. NED will load the component whenever it refreshes or the application starts.

III. A Basic Editor

For a custom editor, it is almost the same process except you extend from the ConfigurationEditorInterface. This requires implementing a couple more methods:

```
public class MyEditor implements ConfigurationEditorInterface
{
public void initialize()
{
        System.out.println("I'm initializing");
}

        public void finish()
{
        System.out.println("I'm done.");
}

public JComponent getEditorComponent()
{
        JPanel panel = new JPanel();
panel.add(new JButton("HI");
return panel;
}

public void update()
{
        System.out.println("I'm updating");
}
}
```

The two new methods are getEditorComponent and update. The getEditorComponentMethod returns whatever SWING component you want to display. This component will show up inside the NED configuration view panel if it is assigned this editor. The update method should handle refreshing your display.

The XML descriptor fields are the same as for a component. However, instead of placing your component in the NED client component directory you place it in the NED editor directory. NED will automatically load the editor component on startup.

To have NED use your editor, create a Group object in NED. If you right click to edit the group properties you will see an “editor” property. If your component loaded properly, you should be able to see your editor in the drop down list. Once you select the editor, click ok, then go and click on the Group. Your editor should display.

IV. Server Interaction

So far, you’ve seen how to create a component and an editor. However, these only operate on client side information. What if you want to do something like download a file from the server for your editor?

In order to have interaction with the server, the server will need a component that knows how to respond to messages from your component. Fortunately this is fairly straightforward as well, and is somewhat similar to creating a client side component.

IV. Basic Server Component

First derive a new class from AbstractServerComponent. This is class contains just one abstract method called registerForMessages. A basic server component may look like this:

```
public class MyServerComponent extends AbstractServerComponent
{
/**
* Registers this component for receiving messages. This method is class specific
        */
public void registerForMessages(ServerCommunicator com)
{
        com.addListener(new MyListener());
}
}
```

The registerForMessages method simply adds all the listeners this component requires for handling messages coming from the client.

IV. Basic Server Message Listener

Now for the listener class:

```
public class MyListener extends ServerMessageListener
{
```

```
        public MyListener()
{
        super(“MyMessage”);
}

protected Message processMessage(Message msg)
{
        System.out.println(“Got it!”);
        Message response = MessageFactory.createMessage(msg, false);

        response.getArguments().add(“Got it!”);
        return response;

}
}
```

A listener must at least do two things. First, it MUST specify the message type that the listener responds to (hence the call to super). All message types are simply strings, but whatever the string is it must be the same on both the client and the server when used.

Second, a listener MUST ALWAYS RETURN A RESPONSE, even if it is just an empty response. The processMessage method takes an incoming message, handles it, and then returns a response that will eventually sent back to the client.

A message object is a JAXB object capable of holding a message status and any number of strings for passing information. This strings can be just plain information strings, base64 encoded strings (in case you need to send binary), or the string representations of other JAXB objects. It depends on what is needed. For example, the GEOSHistoryComponent sends a History JAXB object to the client by serializing the object to a string and then sending that string in a response message. The client then deserializes the string back into a History JAXB object.

Now that you have a ServerComponent and a ServerMessageListener, a component XML descriptor needs to be created. This is very similar to the client component descriptor:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ServerComponentConfiguration xmlns="NED">
        <Name>My Component</Name>
        <Description></Description>
        <Version>1.0.0.0</Version>
        <ClassName>MyComponent</ClassName>
        <Type>JAVA</Type>
        <LinkedComponents></LinkedComponents>
</ServerComponentConfiguration>
```

Now package everything into a jar put it into the NED Server component directory. Once the server is restarted your component will be available to handle messages.

V. Basic Client Side Communication

The last part of getting client/server interactivity is creating a client side interface for sending and receiving your messages. The interface can either be a class (if more than one component may need access to it) or can simply be method calls within your client component.

A basic method for communicating with a server component would be something like this:

```
public void sendMyMessage(String message)
{
Message msg = MessageFactory.createMessage("MyMessage", message);

        //this is a synchronous call
        Message resp = NEDClientConnectionManager.getInstance().sendSyncMessage(msg);

        if (resp != null && (resp.getArguments() != null && resp.getArguments().size() > 0))
        {
                System.out.println(resp.getArguments().get(0));
}
}
```

This method creates a message of type "MyMessage" with a string message, sends it synchronously to the server, and prints the response (if there is one).

IV. The End?

NED has a lot of support classes and synchronous and asynchronous communication. The server is capable of both push and pull operations as well. The code is fairly well documented though I'm sure there are places where it may have gotten out of date or not updated. There may also be places that could be implemented better or refactored. Feel free to update things as you go along.

Also, look at existing components for more "advanced" ways of component development.